Java programming

DB2 Information Management Software

http://www-136.ibm.com/developerworks/db2

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

What is this tutorial about?

In this tutorial, you'll learn about writing Java code that interfaces with DB2 Universal Database. You'll see how to:

- Connect to a DB2 database from a Java application
- Use JDBC to read and update data in a DB2 database
- Use SQLJ to read and update data in a DB2 database
- Troubleshoot a DB2 Java application

This tutorial provides you with the fundamental skills required to develop Java applications for DB2.

This is the fifth in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 5 of the exam, entitled "Java programming." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 26) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

This tutorial is one of the tools that can help you prepare for Exam 703. You should also review Resources on page 26 at the end of this tutorial for more information.

In addition to this DB2 background, you should also have basic knowledge of SQL and the Java platform before taking this tutorial.

System requirements

If you plan to develop Java applications for DB2, or you plan to try out this tutorial's sample code yourself, you need to complete the following tasks:

- Install DB2 Universal Database, Version 8.1 for Linux, UNIX, or Windows.
- Create the sample database by running `db2sampl` from the DB2 CLP. The sample code in this article has been written to work with this database.
- Install the Java Runtime Environment, Java 2 Technology Edition, Version 1.3.1.
- Update the `PATH` environment variable to include the `jdk\bin` directory of the Java SDK you have installed. Note that DB2 includes Java SDK, version 1.3.1, in the `$DB2PATH\sqllib\java\jdk\bin` directory (`$DB2PATH/sqllib/java/jdk/bin` in UNIX).

About the author

Dirk deRoos (BA, BCS) is a technical writer and samples developer on the DB2 Information Development team. He has recently coauthored *The Official Guide to DB2 Version 8.1.2* (Prentice-Hall, 2003), and wrote chapters for *DB2: The Complete Reference* (Osborne/McGraw-Hill, 2001). Dirk specializes in database performance monitoring and DB2 application development. He is a DB2 Certified Solutions Expert (Application Development, Business Intelligence).

You can reach Dirk by sending a message to *dderoos_ibm at yahoo.ca* (mailto:dderoos_ibm@yahoo.ca) .

Notices and trademarks

Section 2. Connecting to a database

Overview

JDBC is a Java API through which Java programs can connect to relational databases and execute SQL statements. To enable Java applications to communicate with its databases, DB2 includes JDBC drivers.

In this section, you'll learn how to load the most appropriate JDBC driver for your application, and how to connect to a DB2 database.

JDBC drivers

A JDBC driver acts as an interface between a JDBC program and a database. DB2 includes three JDBC drivers: the DB2 JDBC Type 2 Driver, the DB2 JDBC Type 3 Driver, and the DB2 Universal JDBC Driver.

DB2 JDBC Type 2 Driver

This JDBC driver is known also as the *app driver.* Java applications that use this driver must run on a DB2 client, through which JDBC requests flow to a DB2 server. This driver will be deprecated in DB2 UDB Version 8.2. It is recommended that you use the DB2 Universal JDBC Driver, which was introduced in DB2 UDB Version 8.1.2, instead of the DB2 JDBC Type 2 Driver.

To load the DB2 JDBC Type 2 Driver, invoke the `Class.forName()` method with `COM.ibm.db2.jdbc.app.DB2Driver` as an argument.

DB2 JDBC Type 3 Driver

This JDBC driver is also known as the *applet* or *net* driver. You can only use the JDBC Type 3 Driver to create Java applets. When your applet calls the JDBC API to connect to DB2, the JDBC driver establishes a separate network connection with the DB2 database through the JDBC applet server residing on the Web server. This driver is deprecated in DB2 UDB Version 8.1, but is provided for backward compatibility. Do not develop new applications that use this driver.

To load this driver, invoke the `Class.forName()` method with `COM.ibm.db2.jdbc.net.DB2Driver` as an argument.

DB2 Universal JDBC Driver

The DB2 Universal JDBC Driver, available in DB2 UDB Version 8.1.2, provides both Type 2 and Type 4 connectivity. Thus, you can use this driver for both applets and applications. In fact, it is the recommended driver for both.

To load the DB2 Universal JDBC Driver, invoke the `Class.forName()` method with `com.ibm.db2.jcc.DB2Driver` as an argument.

To use any of these JDBC drivers in your application, you need to import the Java packages that contain the JDBC API:

```
import java.sql.*;
```

All of the code examples and sample applications in this tutorial have been tested to work with both the DB2 JDBC Type 2 Driver and the DB2 Universal JDBC Driver.

Making a database connection

Once you've loaded the appropriate JDBC driver, you can connect to a database from your JDBC application. In JDBC applications, a database connection is represented by a `Connection` object. From a `DriverManager` object (available for use once the JDBC driver has been loaded), you can use the `getConnection()` method to acquire a `Connection`. In the following example, the DB2 Universal JDBC Driver is loaded, and the `DriverManager` object generates a Type 2 database connection.

```
String url = "jdbc:db2:sample";
Class.forName("com.ibm.db2.jcc.DB2Driver");
Connection con = DriverManager.getConnection(url);
```

The DB2 JDBC Type 2 Driver supports the above URL format, while the DB2 JDBC Type 3 Driver does not.

For the DB2 Universal JDBC Driver, the type of database connection is determined by the URL passed to the `DriverManager.getConnection()` method. In URLs such as the following, where a domain name and port number of the database server are listed, the `DriverManager` object will generate a Type 4 database connection:

```
String url = "jdbc:db2:sample://localhost:5021";
```

The DB2 JDBC Type 3 Driver also supports the above URL format, while the DB2 JDBC Type 2 Driver does not.

Database connection: Sample code

The following application demonstrates all the concepts discussed in this section:

- Importing the Java packages that contain the JDBC API
- Loading the DB2 Universal JDBC Driver
- Creating a `Connection` object
- Using the `DriverManager` to open a `Connection`

This application represents a shell of a typical JDBC application: The JDBC packages are imported, there is a database connection, and there is error handling logic in the `try`/`catch` blocks. To demonstrate that a database connection is actually made, the `Connection` object requests the name of the JDBC driver being used with the `Connection.getMetaData().getDriverName()` methods.

```
//ConnDb.java
import java.sql.*;

class ConnDb
{
  public static void main(String[] argv)
  {
    String url = "jdbc:db2:sample";
    Connection con;

    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      con = DriverManager.getConnection (url);

      System.out.println("JDBC driver name: " +
         con.getMetaData().getDriverName());

      con.close();
    }
    catch (ClassNotFoundException drvEx)
    {
      System.err.println("Could not load JDBC driver");
      System.out.println("Exception: " + drvEx);
      drvEx.printStackTrace();
    }
    catch(SQLException sqlEx)
    {
      while(sqlEx != null) {
        System.err.println("SQLException information");
        System.err.println("Error msg: " + sqlEx.getMessage());
        System.err.println("SQLSTATE: " + sqlEx.getSQLState());
        System.err.println("Error code: " + sqlEx.getErrorCode());
        sqlEx.printStackTrace();
        sqlEx=sqlEx.getNextException();
      }
    }
  }
}
```

To compile the above file (named `ConnDb.java`), execute the following command:

```
javac ConnDb.java
```

To run the compiled application, execute the following command:

```
java ConnDb
```

Here's what the application's output should look like:

```
JDBC driver name: IBM DB2 JDBC 2.0 Type 2
```

Section 3. Reading and updating data with JDBC

Overview

In this section, you'll learn how to use the `Statement` and `PreparedStatement` JDBC objects, which represent SQL statements in JDBC. You'll also learn how to use the JDBC `ResultSet` object, which is returned by `Statement` and `PreparedStatement` objects that contain SQL queries.

The Statement object

`Statement` objects are created using the `Connection.createStatement()` method. For example:

```
Statement stmt;
...
stmt = con.createStatement();
```

To execute an `INSERT`, `UPDATE`, or `DELETE` statement from a `Statement` object, pass a string with the statement to the `Statement.executeUpdate()` method. For example:

```
stmt.executeUpdate("DELETE FROM EMPLOYEE WHERE EMPNO = '000099'");
```

To execute a query using a `Statement` object, pass a string with the `SELECT` statement to the `Statement.executeQuery()` method, and retrieve the `ResultSet` object. For example:

```
ResultSet rs;
rs = stmt.executeQuery("SELECT EMPNO, LASTNAME FROM EMPLOYEE");
```

To parse a `ResultSet` object, you must first fetch each row using the `ResultSet.next()` method. Then, after each fetch, retrieve the column values using the methods applicable to the data type (for instance, `ResultSet.getInt()`).

```
rs = stmt.executeQuery("SELECT LASTNAME, BIRTHDATE FROM EMPLOYEE");
while (rs.next()) {
  System.out.println(rs.getString(1) + ", " + rs.getDate(2));
}
```

Statement object: Sample code

The following application demonstrates all the concepts discussed in the previous panel:

- Creating a `Statement` object
- Executing an `INSERT`, `UPDATE`, or `DELETE` statement
- Executing an SQL query
- Parsing a `ResultSet` object

This application inserts an employee record into the employee table, and then runs a query against that table, requesting the employee number and last name for every record.

```
//StmtDb.java
import java.sql.*;

class StmtDb
{
  public static void main(String[] argv)
  {
    String url = "jdbc:db2:sample";
    Connection con;
    Statement stmt1, stmt2;
    ResultSet rs;
    String stmt1String =
       "INSERT INTO EMPLOYEE (EMPNO,FIRSTNME,MIDINIT,LASTNAME,EDLEVEL)" +
       " VALUES ('000099','MICHIEL','G','DEROOS',1)";
    String stmt2String = "SELECT EMPNO, LASTNAME FROM EMPLOYEE";

    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      con = DriverManager.getConnection(url);
      con.setAutoCommit(true);

      stmt1 = con.createStatement();
      stmt2 = con.createStatement();

      stmt1.executeUpdate(stmt1String);
      rs = stmt2.executeQuery(stmt2String);

      System.out.println("Employee #  Employee surname");
      while (rs.next()) {
        System.out.println(rs.getString(1) + "      " + rs.getString(2));
      }

      rs.close();
      stmt1.close();
      stmt2.close();
      con.commit();
      con.close();

    }
    catch (ClassNotFoundException drvEx)
    {
```

```
        System.err.println("Could not load JDBC driver");
        System.out.println("Exception: " + drvEx);
        drvEx.printStackTrace();
      }
      catch(SQLException sqlEx)
      {
        while(sqlEx != null) {
          System.err.println("SQLException information");
          System.err.println("Error msg: " + sqlEx.getMessage());
          System.err.println("SQLSTATE: " + sqlEx.getSQLState());
          System.err.println("Error code: " + sqlEx.getErrorCode());
          sqlEx.printStackTrace();
          sqlEx=sqlEx.getNextException();
        }
      }
    }
  }
```

To compile the above file (named `StmtDb.java`), execute the following command:

```
javac StmtDb.java
```

To run the compiled application, execute the following command:

```
java StmtDb
```

Here's what the output of this application should look like:

```
Employee #  Employee surname
000010      HAAS
000020      THOMPSON
000030      KWAN
000050      GEYER
000060      STERN
000070      PULASKI
000090      HENDERSON
000100      SPENSER
000110      LUCCHESSI
000120      O'CONNELL
000130      QUINTANA
000140      NICHOLLS
000150      ADAMSON
000160      PIANKA
000170      YOSHIMURA
000180      SCOUTTEN
000190      WALKER
000200      BROWN
000210      JONES
000220      LUTZ
000230      JEFFERSON
000240      MARINO
000250      SMITH
000260      JOHNSON
000270      PEREZ
000280      SCHNEIDER
000290      PARKER
```

```
000300      SMITH
000310      SETRIGHT
000320      MEHTA
000330      LEE
000340      GOUNOT
000099      DEROOS
```

The PreparedStatement object

`PreparedStatement` objects are created using the `Connection.prepareStatement()` method. For example:

```
PreparedStatement stmt;
...
stmt = con.prepareStatement();
```

With a `PreparedStatement` object, you can dynamically prepare and execute SQL statements. By rebinding values to parameter markers in SQL statements, you can execute the same statement multiple times using different values. (A parameter marker is represented by a ?, and acts as a placeholder for a value to be assigned at run time. For example:

```
pStmt = con.prepareStatement("UPDATE STAFF SET COMM=? WHERE ID=?");
pStmt.setDouble(1,710.53);
pStmt.setInt(2,350);
pStmt.executeUpdate();

pStmt.setDouble(1,710.53);
pStmt.setInt(2,350);
pStmt.executeUpdate();
```

For more on parameter markers, read the sixth article in this series(see Resources on page26).

PreparedStatement object: Sample code

The following application demonstrates all the concepts discussed in the previous panel:

- Creating a `PreparedStatement` object
- Using parameter markers to dynamically assign values in SQL statements

This application parses through an array of names and job titles. For each array entry, it inserts a record in the staff table.

```
//pStmtDb.java
import java.sql.*;

class pStmtDb
{
  public static void main(String[] argv)
  {
    String url = "jdbc:db2:sample";
    Connection con;
    PreparedStatement pStmt;
    Statement sStmt;
    ResultSet rs;
    String stmtString = "INSERT INTO STAFF (ID,NAME,DEPT,JOB) VALUES (?,?,99,?)";
    String [][] staff = {{"Smyth","LW"},{"Hemsky","RW"},{"York","C"}};

    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      con = DriverManager.getConnection(url);
      con.setAutoCommit(false);

      pStmt = con.prepareStatement(stmtString);

      for (int i=0; i<3; i++) {
        pStmt.setInt(1,401+i);
        pStmt.setString(2,staff[i][0]);
        pStmt.setString(3,staff[i][1]);
        pStmt.executeUpdate();
      }

      con.commit();

      sStmt = con.createStatement();
      rs = sStmt.executeQuery("SELECT ID, NAME FROM STAFF WHERE DEPT = 99");

      System.out.println("Employee #  Employee surname");
      while (rs.next()) {
        System.out.println(rs.getString(1) + "         " + rs.getString(2));
      }

      rs.close();
      pStmt.close();
      sStmt.close();
      con.commit();
      con.close();
    }
    catch (ClassNotFoundException drvEx)
    {
      System.err.println("Could not load JDBC driver");
      System.out.println("Exception: " + drvEx);
      drvEx.printStackTrace();
    }
    catch(SQLException sqlEx)
    {
      while(sqlEx != null) {
        System.err.println("SQLException information");
        System.err.println("Error msg: " + sqlEx.getMessage());
        System.err.println("SQLSTATE: " + sqlEx.getSQLState());
        System.err.println("Error code: " + sqlEx.getErrorCode());
        sqlEx.printStackTrace();
        sqlEx=sqlEx.getNextException();
      }
    }
```

```
  }
}
```

To compile the above file (named `pStmtDb.java`), execute the following command:

```
javac pStmtDb.java
```

To run the compiled application, execute the following command:

```
java pStmtDb
```

The output of this application should look like this:

```
Employee #  Employee surname
401         Smyth
402         Hemsky
403         York
```

Committing transactions

In the `Statement` and `PreparedStatement` code samples, two different approaches to committing transactions were used: *autocommit* and *manual commit*.

The approach used in the `Statement` sample was autocommit, where each statement was automatically committed. Autocommit is enabled from the `Connection` object as follows:

```
con.setAutoCommit(true);
```

The approach used in the `PreparedStatement` sample was manual commit. With this approach, each statement is either manually committed or rolled back. If statements are not committed by the end of the application, they are automatically rolled back. Manual commit or rollback operations are performed from the `Connection` object as follows:

```
con.commit();
...
con.rollback();
```

Issuing distributed transactions

To issue distributed transactions from your Java applications, you need to use the Java Transaction API (JTA). (Distributed transactions, also referred to as two-phase commit transactions, are transactions that update data in more than one database.) The JTA specification is the transaction management component of the Java 2 Platform, Enterprise Edition (J2EE) standard.

In DB2, distributed transactions are managed through the `DB2XADataSource` class, which implements the `XADataSource` interface from the `javax.sql` package.

For DB2 UDB Version 8.1, JTA support is only provided in the DB2 JDBC Type 2 Driver. The following Java code will create an instance of the `DB2XADataSource` class using the DB2 JDBC Type 2 Driver:

```
DB2XADataSource db2ds = new COM.ibm.db2.jdbc.DB2XADataSource();
```

In DB2 UDB Version 8.2, the DB2 Universal JDBC Driver will support the JTA. The following Java code will create an instance of the `DB2XADataSource` class using the DB2 Universal JDBC Driver:

```
DB2XADataSource db2ds = new com.ibm.db2.jcc.DB2XADataSource();
```

Section 4. Reading and updating data with SQLJ

Overview

The SQLJ API is an extension of JDBC, and it supports the static execution of SQL statements. Because DB2 supports SQLJ, Java developers can overcome a major limitation of JDBC, namely that JDBC can only execute SQL statements dynamically. Statically bound SQL statements typically run faster than dynamically bound statements, which gives SQLJ applications a significant performance advantage over JDBC applications.

In this section, you'll learn how to program with the SQLJ API by using connection contexts, issuing SQL statements in SQLJ executable clauses, and parsing result sets with iterators.

Tasks for developing SQLJ applications

At the beginning of an SQLJ application, you need to import the Java packages that contain the JDBC and SQLJ APIs:

```
import sqlj.runtime.*;
import java.sql.*;
```

SQLJ applications use the same JDBC drivers discussed in JDBC drivers on page 4.

Before SQLJ source code can be compiled, it needs to be transformed into Java code so that it can be compiled by a Java compiler. The `sqlj` command performs this transformation and also invokes the Java compiler. If you have SQL statements in your SQLJ source file, a file with the ending `_SJProfile0.ser` will appear in your directory alongside the `class` file and `sqlj` file.

You can run your source code once you've translated and compiled it. However, the SQL statements in your application will not be statically bound, potentially resulting in poor performance. To statically bind the SQL statements in your application, you need to perform a *profile customization* of your application. You can perform this customization with the `db2sqljcustomize` tool. (This tool was formerly known as `db2profc`. As of DB2 Version 8.1, `db2profc` is deprecated.)

To perform a profile customization of this compiled application, execute the following command:

```
db2sqljcustomize -url jdbc:db2://localhost:50000/sample
                 -user uid -password pwd
```

```
                    app_SJProfile0.ser
```

In this command, `uid` represents your user ID, `pwd` represents your password, and `app` represents the name of your application.

Here is some sample output from a successful run of `db2sqljcustomize`:

```
db2sqljcustomize -url jdbc:db2://localhost:50000/sample
                 -user me -password mypwd DbApp_SJProfile0.ser
[ibm][db2][jcc][sqlj]
[ibm][db2][jcc][sqlj] Begin Customization
[ibm][db2][jcc][sqlj] Loading profile: DbApp_SJProfile0
[ibm][db2][jcc][sqlj] Customization complete for profile DbApp_SJProfile0.ser
[ibm][db2][jcc][sqlj] Begin Bind
[ibm][db2][jcc][sqlj] Loading profile: DbApp_SJProfile0
[ibm][db2][jcc][sqlj] Binding package DBAPP01 at isolation level UR
[ibm][db2][jcc][sqlj] Binding package DBAPP02 at isolation level CS
[ibm][db2][jcc][sqlj] Binding package DBAPP03 at isolation level RS
[ibm][db2][jcc][sqlj] Binding package DBAPP04 at isolation level RR
[ibm][db2][jcc][sqlj] Bind complete for DbApp_SJProfile0
```

Establishing a connection context

In SQLJ applications, SQL operations require a *connection context* instead of a JDBC `Connection` object. Each SQL statement is issued through `#sql` SQLJ clauses, and is explicitly or implicitly assigned a connection context. The assignment of a connection context associates the statement with a previously opened database connection.

Before establishing a database connection, you must first generate a connection context class. The following line of code generates a connection context class named `Ctx`:

```
#sql context Ctx;
```

To establish a database connection, load a JDBC driver using the `Class.forName()` method, and then construct an instance of the context class you generated earlier. At minimum, you need to pass the context constructor the URL of the database and a boolean indicating whether or not you want autocommit turned on. The following code snippet establishes a database connection in SQLJ:

```
String url = "jdbc:db2:sample";
Class.forName("com.ibm.db2.jcc.DB2Driver");
Ctx connCtx = new Ctx(url,true);
```

In our sample connection context `connCtx`, the DB2 Universal JDBC Driver is being used to connect to the sample database on a DB2 server, and

autocommit is on.

Issuing SQL statements

A significant advantage of SQLJ over many other database APIs (including JDBC) is the simplicity and power of SQLJ's syntax. The following code example demonstrates an UPDATE statement that makes use of a host variable:

```
String staffJob = "Sales";
#sql [connCtx] {UPDATE STAFF SET COMM=400 WHERE JOB = :staffJob};
```

To write SQL statements that return result sets, you'll need to use *iterators*. Iterators are discussed later in Using iterators on page 18.

Basic SQLJ application: Sample code

The following application demonstrates all the concepts discussed in the previous three panels:

- Importing the Java packages that contain the JDBC and SQLJ APIs
- Loading the DB2 Universal JDBC Driver
- Generating a connection context
- Executing an SQL statement with a host variable

In this application, a SELECT statement is executed, and the scalar result is passed to a host variable named `stname`.

```
//DbApp.sqlj
import sqlj.runtime.*;
import java.sql.*;

#sql context Ctx;

class DbApp
{
  public static void main(String[] argv)
  {
    String url = "jdbc:db2:sample";
    String stname;

    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");

      Ctx connCtx = new Ctx(url,true);

      #sql [connCtx] {SELECT NAME INTO :stname FROM STAFF WHERE ID=10};
```

```
        System.out.println("The name of employee #10 is: " + stname);

        connCtx.close();
      }
      catch (ClassNotFoundException drvEx)
      {
        System.err.println("Could not load JDBC driver");
        System.out.println("Exception: " + drvEx);
        drvEx.printStackTrace();
      }
      catch(SQLException sqlEx)
      {
        while(sqlEx != null) {
          System.err.println("SQLException information");
          System.err.println("Error msg: " + sqlEx.getMessage());
          System.err.println("SQLSTATE: " + sqlEx.getSQLState());
          System.err.println("Error code: " + sqlEx.getErrorCode());
          sqlEx.printStackTrace();
          sqlEx=sqlEx.getNextException();
        }
      }
    }
  }
```

To translate and compile the above file (named `DbApp.sqlj`), execute the following command:

```
sqlj DbApp.sqlj
```

To perform a profile customization of this compiled application, execute the following command:

```
db2sqljcustomize -url jdbc:db2://localhost:50000/sample
                 -user uid -password pwd DbApp_SJProfile0.ser
```

In this command, `uid` represents your user ID, and pwd represents your password.

To run the compiled application, execute the following command:

```
java DbApp
```

The output of this application should look like this:

```
The name of employee #10 is: Sanders
```

Using iterators

To parse result sets in SQLJ applications, you need to use iterators. Iterators

are the SQLJ version of SQL cursors. There are two main kinds of iterators you can use in SQLJ: *named* iterators and *positional* iterators. We'll discuss each in more detail in the next two panels.

Named iterators

Named iterators are declared with the names and data types of the columns in the result set. With named iterators, the order of the columns does not matter. Before you can use a named iterator in your application, you must generate an iterator class, keeping in mind the nature of the result set you want to parse. The following code snippet generates a named iterator class called `NameIter`; the iterator has two columns.

```
#sql iterator NameIter(String Name, int Id);
```

When issuing an SQL query, you should pass the result set to an instance of the iterator class you generated. For example:

```
NameIter nIter;
#sql [connCtx] nIter = {SELECT ID, NAME FROM STAFF WHERE DEPT = 20};
```

To retrieve column values from a named iterator, you can use the iterator methods named after the column names in the result set. To parse the named iterator, you use a loop structure; before each loop cycle, go to the next row by running the `iterator.next()` method. The following code snippet parses an instance of the `NameIter` named iterator.

```
while (nIter.next()) {
  System.out.println(nIter.Name() + ", ID #" + nIter.Id());
}
```

Positional iterators

Positional iterators are declared only with the data types of the columns in the result set. With positional iterators, the order of the columns does matter. Before you can use a positional iterator in your application, you must generate an iterator class, keeping in mind the nature of the result set you want to parse. The following code snippet generates a positional iterator class called `PosIter`; the iterator has two columns.

```
#sql iterator PosIter(String, int);
```

When issuing an SQL query, pass the result set to an instance of the iterator

class you generated. For example:

```
PosIter pIter;
#sql [connCtx] pIter = {SELECT ID, NAME FROM STAFF WHERE DEPT = 20};
```

To retrieve column values from a positional iterator, you need to fetch those values into host variables. Use the `iterator.endFetch()` method to determine if there are additional rows to fetch. The following code snippet parses an instance of the `PosIter` positional iterator.

```
#sql {FETCH :pIter INTO :nameHv, :idHv };
while (!pIter.endFetch()) {
  System.out.println(nameHv + ", ID #" + idHv);
  #sql {FETCH :pIter INTO :nameHv, :idHv };
}
```

Named and positional iterators: Sample code

The following application demonstrates all the concepts discussed in the previous three panels:

- Generating a named iterator
- Generating a positional iterator
- Parsing a result set using a named iterator
- Parsing a result set using a positional iterator

In this application, a `SELECT` statement is executed twice. The result set from the first execution is parsed with a named iterator called `nIter`. The result set from the second execution is parsed with a positional iterator called `pIter`.

```
//Iter.sqlj
import sqlj.runtime.*;
import java.sql.*;

#sql context Ctx;
#sql iterator NameIter(String Name, int Id);
#sql iterator PosIter(String, int);

class Iter
{
  public static void main(String[] argv)
  {
    String url = "jdbc:db2:sample";
    String nameHv = null;
    int idHv = 0;
    NameIter nIter;
    PosIter pIter;

    try
    {
      Class.forName("com.ibm.db2.jcc.DB2Driver");
```

```
      Ctx connCtx = new Ctx(url,true);

      System.out.println("\nResult set from named iterator:");
      #sql [connCtx] nIter = {SELECT ID, NAME FROM STAFF WHERE DEPT = 20};
      while (nIter.next()) {
        System.out.println(nIter.Name() + ", ID #" + nIter.Id());
      }
      nIter.close();

      System.out.println("\nResult set from positional iterator:");
      #sql [connCtx] pIter = {SELECT NAME, ID FROM STAFF WHERE DEPT = 20};
      #sql {FETCH :pIter INTO :nameHv, :idHv };
      while (!pIter.endFetch()) {
        System.out.println(nameHv + ", ID #" + idHv);
        #sql {FETCH :pIter INTO :nameHv, :idHv };
      }
      pIter.close();

      connCtx.close();
    }
    catch (ClassNotFoundException drvEx)
    {
      System.err.println("Could not load JDBC driver");
      System.out.println("Exception: " + drvEx);
      drvEx.printStackTrace();
    }
    catch(SQLException sqlEx)
    {
      while(sqlEx != null) {
        System.err.println("SQLException information");
        System.err.println("Error msg: " + sqlEx.getMessage());
        System.err.println("SQLSTATE: " + sqlEx.getSQLState());
        System.err.println("Error code: " + sqlEx.getErrorCode());
        sqlEx.printStackTrace();
        sqlEx=sqlEx.getNextException();
      }
    }
  }
}
```

To translate and compile the above file (named `Iter.sqlj`), execute the following command:

```
sqlj Iter.sqlj
```

To perform a profile customization of this compiled application, execute the following command:

```
db2sqljcustomize -url jdbc:db2://localhost:50000/sample
                 -user uid -password pwd Iter.ser
```

In this command, `uid` represents your user ID, and pwd represents your password.

To run the compiled application, execute the following command:

```
java Iter
```

The output of this application should look like this:

```
Result set from named iterator:
Sanders, ID #10
Pernal, ID #20
James, ID #80
Sneider, ID #190

Result set from positional iterator:
Sanders, ID #10
Pernal, ID #20
James, ID #80
Sneider, ID #190
```

Section 5. Troubleshooting

Overview

In this section, you'll learn how to use the troubleshooting tools available for JDBC applications. We'll examine basic error handling and reporting practices using `SQLException` objects; we'll use the JDBC trace facility; and we'll discuss the JDBC error log.

Error handling

If you follow proper error handling practice in your Java programs, you wrap the logic in your JDBC applications in `try`/`catch` blocks. When errors occur in JDBC methods, they throw `SQLException` objects. Therefore, for every `try` block that contains a JDBC operation, the corresponding `catch` block should contain logic to handle an `SQLException` object.

The following example features a `SELECT` statement with type-incompatible operands in the `WHERE` clause: the `JOB` column of the `STAFF` table is of data type `VARCHAR`, not `INTEGER`.

```
try {
...
rs = stmt.executeQuery("SELECT ID, NAME FROM STAFF WHERE JOB = 99");
...
}
catch {
  while(sqlEx != null) {
    System.err.println("SQLException information");
    System.err.println("Error msg: " + sqlEx.getMessage());
    System.err.println("SQLSTATE: " + sqlEx.getSQLState());
    System.err.println("Error code: " + sqlEx.getErrorCode());
    sqlEx=sqlEx.getNextException();
  }
}
```

The `catch` block contains a `while` loop, which facilitates the handling of multiple exceptions. At the end of the loop, there is a call of the `SQLException.getNextException()` method, which returns an exception if there is another exception to catch, or null if there are no additional exceptions.

The `SELECT` statement with the incompatible operands generates the following exceptions:

```
SQLException information
Error msg: DB2 SQL error: SQLCODE: -401, SQLSTATE: 42818, SQLERRMC: =
SQLSTATE: 42818
Error code: -401
```

```
SQLException information
Error msg: DB2 SQL error: SQLCODE: -727, SQLSTATE: 56098, SQLERRMC: 2;-401;42818;=
SQLSTATE: 56098
Error code: -727
```

This output will differ depending on the JDBC driver you are using. The above output was generated by an application using the DB2 Universal JDBC driver.

All the complete code samples presented in this tutorial contain logic to handle `SQLException` objects.

JDBC trace

To debug your JDBC applications, you can use the JDBC trace facility. However, you can only use this facility for applications that use the DB2 JDBC Type 2 Driver.

To activate the JDBC trace facility, add the following lines to the `db2cli.ini` file found in the `sqllib` directory:

```
[COMMON]
JDBCTrace=1
JDBCTraceFlush=1
JDBCTracePathName="C:\temp"
```

For the `JDBCTracePathName` keyword, you must refer to an existing directory path.

Once you have activated the JDBC trace facility, every time you run an application that uses the DB2 JDBC Type 2 Driver, a file is created in the `JDBCTracePathName` directory with a detailed trace of every JDBC call. Here is an example of some JDBC trace data generated from the `StmtDb.java` application developed in Statement object: Sample code on page 9:

```
jdbc.app.DB2Driver -> DB2Driver() (2004-03-02 12:00:45.406)
| Loaded db2jdbc from java.library.path
| DB2Driver: JDBC 2.0,  BuildLevel: s030728
...
jdbc.app.DB2Statement -> executeQuery( SELECT EMPNO, LASTNAME FROM EMPLOYEE )
(2004-03-02 12:00:45.797)
```

When you finish your debugging activities, be sure to turn off the JDBC trace facility. When the trace is on, there is significant performance degradation, and log files are continually generated. If the JDBC trace facility is left on, these log files can consume all of your available disk space. To turn off the JDBC trace facility, edit the `JDBCTrace` keyword entry in the `db2cli.ini` file as follows:

```
JDBCTrace=0
```

JDBC error log

The JDBC error log consists of a file called `jdbcerr.log`; it logs the activity of the now deprecated DB2 JDBC Type 3 Driver. `jdbcerr.log` exists in the directory with the other DB2 diagnostic files, such as `db2diag.log`. The location of this directory is determined by the `DIAGPATH` database manager configuration parameter.

Section 6. Conclusion

Summary

This tutorial provides a working introduction to the Java database programming concepts you will be tested on in the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). To reinforce the ideas presented in each of the tutorial sections, do more than simply compile and run the sample code. Make your own modifications and enhancements!

Resources

- For more information on the DB2 UDB V8.1 Family Application Development Certification (Exam 703), see *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education/) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one tutorial in a series of seven to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:
  1. *Database objects and Programming Methods*
  2. *Data Manipulation*
  3. *Embedded SQL Programming*
  4. *ODBC/CLI Programming*
  5. Java Programming
  6. *Advanced Programming*
  7. User-Defined Routines
- Before you take the certification exam (DB2 UDB V8.1 Application Development exam, Exam 703) for which this tutorial was created to help you prepare, you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:
  - DB2 planning
  - DB2 security
  - Accessing DB2 UDB data
  - Working with DB2 UDB data
  - Working with DB2 UDB objects
  - Data concurrency
- Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:

- Server management
- Data placement
- Database access
- Monitoring DB2 activity
- DB2 utilities
- Backup and recovery

The following links will provide more information on developing Java applications for DB2 Universal Database:

- Check out the *IBM DB2 Universal Database Application Development Guide: Programming client applications*.
- Learn more about the *JDBC 3.0 specification*.

Visit *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .